第4章

演算回路に不可欠なインターフェースの設計

―― 2005年1月号付属FPGA基板を使って動作を確認

S. Mori

一般にIPコアは，演算回路本体と，そのほかの回路との接続を行うためのインターフェース回路からなる．演算回路の性能を引き出せるようにシステムに組み込んだり，仕様変更に柔軟に対応できるようにするためには，インターフェース回路部をうまく設計する必要がある．ここでは，演算回路をおもにCPUにつなぐ場合に使うインターフェース回路（レジスタ・インターフェース）の概要と設計時の注意点をまとめた．また，例題として，パソコンと演算回路（乗算器）をRS-232-Cで接続して制御する回路の事例を解説する．なお，本記事で紹介する設計データは本誌ホームページ（http://www.cqpub.co.jp/dwm/）からダウンロードできる．（編集部）

ここでは，演算回路と外界をつなぐインターフェースの設計について解説します．特定のインターフェースを対象とするのではなく，基礎となる一般的な考えかたを整理して説明していきます．また，より理解を深めるため，具体例としてパソコンと演算回路（乗算器）をRS-232-Cで接続して制御する回路について解説します．この回路については，本誌2005年1月号付属FPGA基板を利用して動作確認を行いました（**写真1**）．

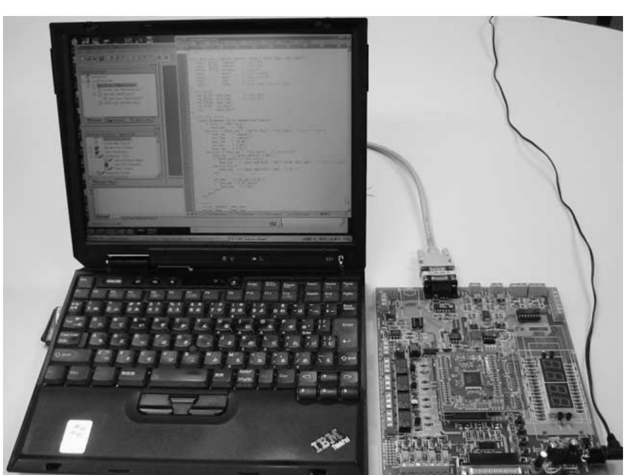


写真1　実機による動作確認のようす

2005年1月号付属FPGA基板とその汎用評価ボード（2005年1月号 特集第6章で紹介）を用いた動作確認．

● 演算回路にはインターフェースが必要不可欠

ところで，「演算回路の設計とインターフェースの話は別に関係ないのでは？」と思う方がいるかもしれません．しかし，演算回路の設計者にとって，インターフェースの知識は避けて通れないものです．

なぜなら，演算回路はかならずほかの回路あるいは機器と接続して使われるからです．演算回路だけを作って「全設計完了！」ということはあまりありません．例えば，積和演算器を使って画像デコーダなどを作ったとき，それを実際に使おうとすると，「入力データを送り，演算をスタートさせ，演算が終わったら出力を取り出す」ためのハードウェアやソフトウェアによるしかけが必要になります．

また，インターフェースのことをまったく考えずに演算回路を設計してしまうと，せっかくの演算性能を発揮できなかったり，最悪の場合，演算回路を組み込めない（ほかの回路と接続できない）といったケースも起こりえます．

このように，演算回路とインターフェースは別物ではなく，切っても切れない関係にあるのです．

● 接続相手が何であれ，インターフェースの基礎は同じ

インターフェースとひと口に言っても，その種類は接続相手の数だけ，ほとんど無数に存在します．例えば，演算回路をシステムLSIに組み込むのであればオンチップ・バス 注1 へのインターフェースが必要ですし，ほかのIPコア

またはPCIなどのバスにつなぐのであればそのためのインターフェースが必要です．それぞれのケースで実装方法がまったく異なるため，本稿ですべてを詳しく説明することはできません．

しかし，接続相手が何であっても，必要な基礎知識はわりと共通しています．そこで，ここではIPコアのインターフェース部分をどのような考えに基づいて設計するかを紹介します．

1 IPコア内に回路の階層を作る

図1に，実用上使いやすい形に構成したIPコアの典型例を示します．例えば，AMBAにつなぐ乗算IPコアを作ろうとしたとき，それを一気に組み立てるのではなく，データの流れに次のような三つの階層を設け，それぞれを回路ブロックとします．

● 接続相手を変更しやすいように階層を作る

まず，乗算などメインの処理を行う回路（**図1**の最下段）ですが，これがいわゆる「生の演算回路」です．本特集をはじめ，さまざまな演算回路の作成記事で扱っているのは，この部分です．

次にCPUと演算回路を仲介するためのレジスタ・インターフェース回路（**図1**の中段）ですが，この回路の役割はCPUと演算回路の間のデータ転送を中継することで，そのためのプロトコル変換（要するにデータ入出力タイミングの変換）を行います．具体的には，まずCPUから受け取った入力データをレジスタにためておき，演算回路へそのデータを送り込みます．次に，演算回路の出力データをレジスタにため，最後にCPUへ渡すという動作を行います．

3番目の外部インターフェース回路（**図1**の最上段）では，例えばAMBAなどのCPUバスであれば，そのバスのデータ転送プロトコル（タイミング・チャート）に従って，レジスタ・インターフェースにあるレジスタの読み書きを行います．

このような回路構成のもとで，データは「CPU→外部（バス）インターフェース→レジスタ・インターフェース→演算回路」という順番に（あるいはその逆の順番に），転送タイミングの変換などを受けつつ移動していきます．バス・インターフェース回路やレジスタ・インターフェース回路は，生の演算回路に対して，ラッパ（wrapper；「くるむもの」という意味）とも呼ばれます．

上記のような階層を設ける理由は，回路変更を容易にするためです．例えば，先述のAMBA用乗算回路があったとして，これをCoreConnectで使おうとしたときに全回路が作り直しになるのではたいへんです．**図1**のような階層を作っておけば，バス・インターフェースの部分だけを差し替えるだけで済みます．

なお，現実に存在するIPコアでは，これほどきれいに回路の階層が分かれておらず，渾然一体となっているものも多く見られます[注2]．しかし，これからIPコアを設計しようという方には，回路規模などの制約がよほど厳しくない限り，なるべく階層を切っておくことをお勧めします．

● 基本構成は昔も今もあまり変わらない

実はこうした回路構成は目新しいものではなく，ひと昔前からあるマイコン・システムとほとんど変わりません（**図2**）．その当時，CPUバスにはいくつかの周辺LSIがぶらさがっていました．周辺LSIの内部には，やはりバス・インターフェースやレジスタ・インターフェースがあって，CPUが

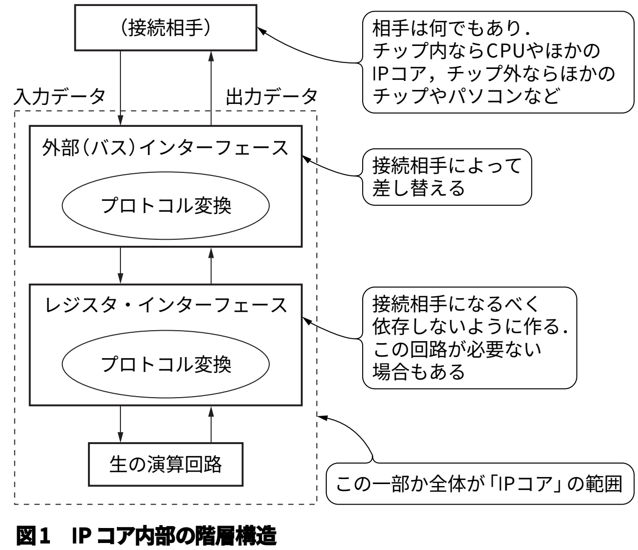


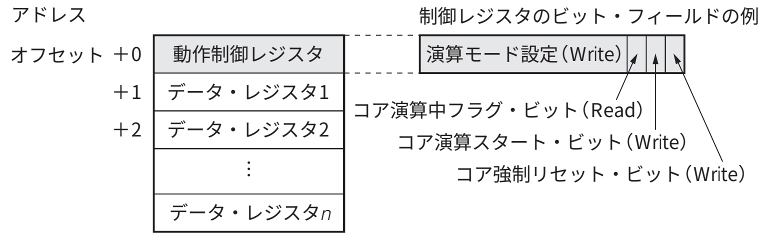
図1　IPコア内部の階層構造

現実には，図に示すほどきれいに内部回路が分離していない場合も多い．しかし，なるべく階層を設けてIPコアを作るようにしておけば，インターフェースを変更するときに回路を全部組み替えなくてもすむし，ほかの設計者への引き継ぎも楽になる．

注1：オンチップ・バスとしては，英国ARM社のAMBA（Advanced Microcontroller Bus Architecture）やOCP-IP（OCP International Partnership）の定めるOCP（Open Core Protocol），米国IBM社のCoreConnectなどが挙げられる．

注2：階層構成は，図1のほかにもいろいろな変形版が考えられる．例えばCPUがなければ，レジスタ・インターフェースはなくてもかまわない．

▶ **図3**
CPUからIPコアを見たときの典型的なレジスタ構成
演算回路をCPUとつなぐ場合，CPUから動きを制御できるようなレジスタ・インターフェースをIPコア内に作る．CPUから見ると，これらのレジスタはアドレス空間の一部にマップされており，該当アドレスを読み書きすることでアクセスできる．ほとんどの場合，レジスタにはデータ入出力用レジスタと動作制御用レジスタの2種類がある．



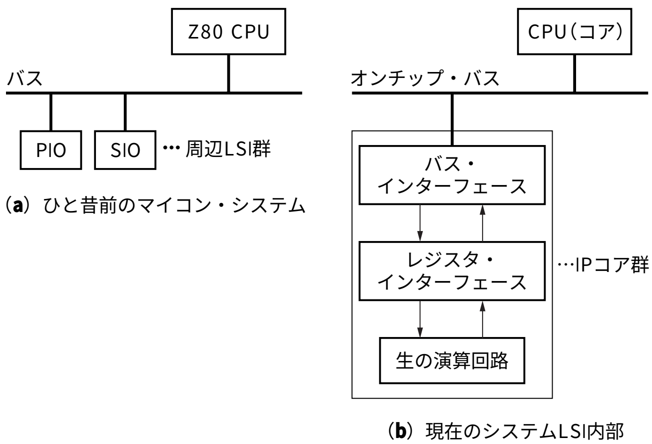
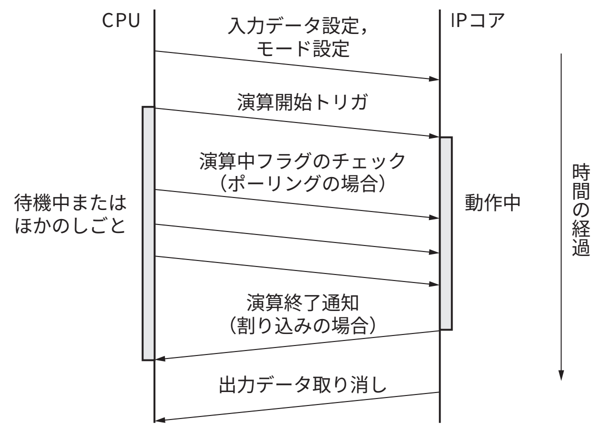


図2　レジスタを使ったデータのやりとりの方法は今も昔も同じ
チップ内部でコアを接続したときに実行することは，昔，CPUと周辺LSIをつなぎ合わせてシステムを組んでいたころと，大筋同じである．レジスタ・インターフェースの構成のしかたもほぼ同じ．



典型的なCPUからのアクセス手順
1）入力データをデータ・レジスタに書く
2）演算の種類などを制御レジスタに設定
3）制御レジスタ経由で，コアに演算開始を指示
4）動作制御レジスタを見ながら，演算終了を待つ（または割り込み）
5）演算結果をデータ・レジスタから読む

図4　CPUとIPコアの間の典型的なやり取り
通常，IPコアの制御レジスタの演算スタート・ビットへアクセスする（例えば'1'を書く）ことで，IPコアの動作が始まる．IPコアの動作終了の検知には，ポーリングと割り込みの2種類の方法がある．ポーリングでは，CPUは「コア演算中フラグ」を読み続け，ほかのしごとはできない．割り込みでは，IPコアがCPUへ演算終了を通知するので，それまでの間，CPUは別のしごとを実行できる．CPUのプログラミングはポーリングのほうが簡単．

レジスタを読み書きすることで周辺LSIにしごとをさせていました．

現在の状況はというと，その昔，複数のICを組み合わせて行っていたことを，チップ内部で実行するようになっています．昔と異なるのは，周辺LSIの中身に相当する部分をIPコア設計者が作らなければならなくなったことと，バスの機能が格段に豊富になってシステム構成も複雑になっていることです．しかし，設計のための基本的な考えかたは，あまり変わっていません．

2 ソフトとハードの接点，レジスタI/F

演算回路をCPUと接続して使うケースは非常に多く，そのときレジスタ・インターフェースが必要になります．まず，典型的な手法を説明し，それからいろいろなバリエーションを紹介します．

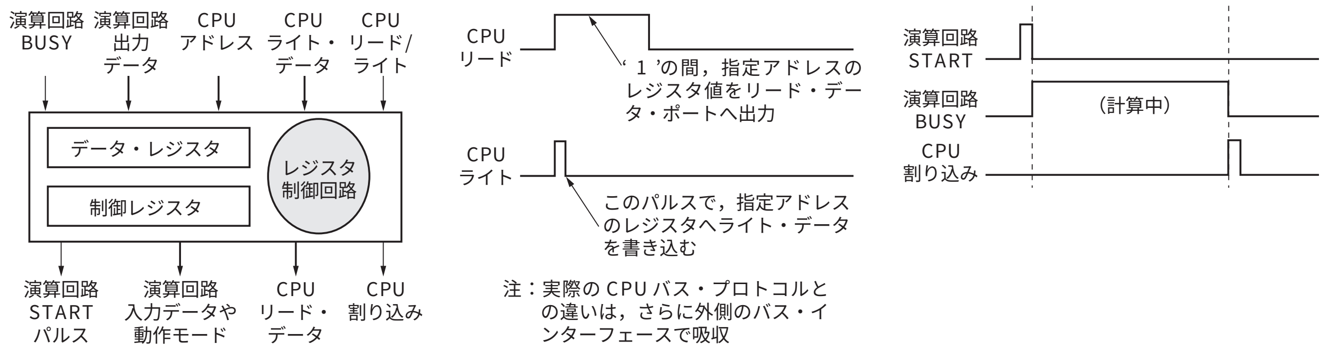
● CPUと演算回路の間のデータをレジスタを介して授受

CPUと演算回路の間でデータをやり取りするために，IPコアの内部にレジスタ（CPU内部のレジスタとは異なる）を設けます．レジスタは，回路としては単なるDフリップフロップ（D-FF）かSRAMです．CPUから見ると，レジスタはメモリ・アドレス空間またはI/Oアドレス空間の一部となります．つまり，ある特定の番地を読み書きすることで，IPコア内のレジスタを読み書きすることができます．

図3に，IPコア内部の典型的なレジスタ構成を示します．ほとんどのIPコアには，大きく分けて次の2種類のレジスタがあります．レジスタの本数やアドレスの振りかたなどは，IPコアによってさまざまです．

- データ・レジスタ：入出力データを置くためのレジスタ
- 制御レジスタ：IPコアの動作を制御するためのレジスタ

多くの場合，上記のようにレジスタを設置したうえで，



（a）レジスタ・インターフェース部のポート　（b）CPUバス側の動作タイミング（SRAMと同様）　（c）演算回路側の動作タイミング

図5　レジスタ・インターフェース回路の基本構成

インターフェースは，演算回路とのデータ入出力，CPUバス側とのデータ入出力，および通信制御信号をポートとして持つ．インターフェース内にある制御回路は，レジスタの読み書き制御信号と，演算回路やCPUバスとの通信制御信号を生成する．

次のようにしてコアを動作させます（**図4**）．

1）IPコアのデータ・レジスタに入力データを書き込む．また，IPコアで行える演算などにいくつか種類がある場合，制御レジスタにモード設定ビットを用意しておき，それを設定する．なお，この時点では，IPコアは待機状態にあるので何も演算処理は行っていない[注3]．

2）CPUがIPコアに演算処理を開始させる．IPコアの制御レジスタに「演算スタート・ビット」を用意しておき，そこへCPUが'1'を書くことでトリガをかける．

3）CPUは，IPコアの処理が終わるまで待機する．待機には，大きく分けて二つの方法がある．一つはポーリング（polling）と呼ばれるもので，IPコアの制御レジスタに「演算中フラグ」を用意しておき，CPUがこれを繰り返し読んでフラグが降りるまで待つという方法．もう一つは，演算終了時に，IPコアのほうからCPUへ割り込み（interrupt）をかける方法．

4）CPUは，IPコアのデータ・レジスタを読んで，演算結果を得る．

上記3）の割り込みの場合，IPコアの動作中にCPUが別の処理を行えます．一方，ポーリングでは，通常CPUは別の処理を行えません．このため，多くの場合は割り込みが使われますが，ポーリングがまったく使われないかというと，そうではありません．ポーリングのほうがCPUソフトウェアのプログラミングが簡単なので，例えば次のようなケースで用いられます．

- CPUがほかの処理を行う必要がない場合
- IPコアの動作時間が短いため，IPコア動作中にCPUがたいしたしごとを行えない場合

● レジスタI/Fの構成

図5（a）に，レジスタ・インターフェース部の基本的な回路構成を示します．インターフェース内部にはデータ・レジスタ，制御レジスタ，およびレジスタ制御回路があります．インターフェースの入出力ポートとしては，大きく分けて次の2種類があります．

1）CPUバス・インターフェースとつなぐポート――入出力データ・バス[注4]とデータ転送制御信号がある．例えば，**図5（b）**のようなタイミング・チャートを想定し，CPUからレジスタ・アドレスやリード/ライト・パルスを受けるとともに，CPUへの割り込み信号を出す．

2）演算回路とつなぐポート――入出力データ・バスと演算回路の動作制御信号がある．**図5（c）**のようなタイミング・チャートを想定し，演算スタート・パルスを出すとともに，BUSY信号を受ける場合が多い．

もちろん，ここで示したのは典型例にすぎず，実際には演算回路やCPUバスのポート，あるいは動作タイミングなどによって入出力ポートは異なります．

ただし一般には，CPUバスの動作タイミングとしては普通のSRAMのようなシンプルなもの（**図5（b）**）を想定して，レジスタ・インターフェース回路を設計しておきます．実際に接続するCPUバスとの動作の違いは，**図1**や**図2**に示した，上位にかぶせるバス・インターフェース回路で吸収

注3：CPUがいつでもコアを待機状態へ戻せるように，IPコアの制御レジスタに「強制リセット」を用意しておくとよい．
注4：通常，チップ内では，CPUの入力用バスと出力用バスは分離する．また，出力用バスについては，有効データを出力していないときに"L"にする（いわゆるORバスと呼ばれるもので，すべてのIPコアの出力バスのORをとり，それをCPUの入力バスにつなぐ）．

します．そして，レジスタ・インターフェース回路では，演算回路の動作タイミングを外界から隠ぺいすることに重点を置くようにします．

図5(a)の回路は，**図4**の各時点に対応して，以下のような動作を行います．

- CPUバス・インターフェースからレジスタへのライト・アクセス（ライト・パルス）が来た場合，レジスタへその値を書き込む
- 制御レジスタの「演算スタート・ビット（**図3**）」に‘1’が書かれた場合，演算回路へスタート・パルスを送る
- 演算が終了して演算回路のBUSY信号が降りたとき，CPUへの割り込み信号を発生するとともに，演算結果をデータ・レジスタへ書き込む
- CPUバス・インターフェースから，レジスタへのリード・アクセスが来た場合，その値をリード・データとして出力する

それでは，レジスタ・インターフェース回路の中身について，もう少し具体的に見ていきましょう．

まず，データ・レジスタの回路構成とHDL記述の骨格部分を**図6**に示します．レジスタ本体はD-FFであり，これに先述の動作に従って読み書きするためのセレクタ，制御信号をつなぎます．もし複数個のデータ・レジスタがあるときは，**図6**の回路が並列に並びます（CPUのリード・データには，レジスタ・アドレスに応じてレジスタ値を選んで出力）．なお，**図6**のHDL記述において，演算回路のBUSY信号が‘1’から‘0’へ変化したことは，**図7**に示すような微分回路によって検出できます．

次に，制御レジスタですが，データ保持が主目的のデータ・レジスタとは構成が異なります（**図8**）．制御レジスタの主な役割は次の二つです．

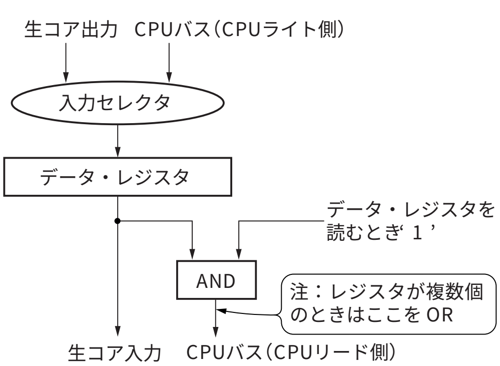
- CPUからライト・アクセスがあったとき，演算回路の動作制御信号を生成する
- CPUからリード・アクセスがあったとき，演算回路の動作状況などをリード・データとして返す

したがって，回路はこれらの信号を生成するためのデコーダ（またはシーケンサ）だけで済みます．動作モードを保持するような場合を除いて，データ・ラッチ用のD-FFはかならずしも必要ではありません[注5]．

● ターゲットに応じたレジスタI/Fのソフトを作る

これまでに説明したレジスタ・インターフェース回路を，バス・インターフェース回路を介してCPUバスにつなぐ

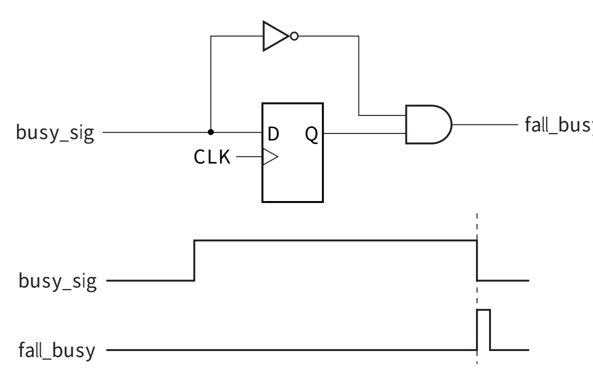
注5：むしろラッチしないほうがよい．もしも，演算回路スタート・ビットなどをD-FFでラッチしてしまうと，CPUが‘1’を書いた後で‘0’に書き戻してもらうか，または自動0クリアするような回路の追加が必要になって，使い勝手が非常に悪い．




```
always @ (posedge clk) begin
    if (レジスタ・アドレス == データ・レジスタのアドレス
            && CPUライト == 1'b1)
        data_reg <= CPUライト・データ;
    else if (演算回路のBUSYが1から0に変化)
        data_reg <= 生コア出力;
end
assign 生コア入力 = data_reg;
assign CPUリード・データ
        = (レジスタ・アドレス == データ・レジスタのアドレス
            && CPUリード == 1'b1) ? data_reg : 0;
```

図6　典型的なデータ・レジスタ回路とVerilog HDL記述例

データ・レジスタは，演算回路出力が得られたときやCPUからのライト・アクセスがあったときに値を取り込む．データ・レジスタの出力は，コア入力とCPUバスへ接続される．




```
always @ (posedge clk) begin
    delay_reg <= busy_sig;
end
assign fall_busy = delay_reg & (~busy_sig);
```

図7　微分回路

微分回路は，信号の変化点を捕らえるためにインターフェース回路ではわりとよく使われる．図では，演算回路からのBUSY信号busy_sigが立ち下がった直後に，1クロック幅のパルスfall_busyが出る．

と，IPコアの中のレジスタをCPUのアドレス空間にマッピングできます．このとき，CPUからどのようにIPコアへアクセスするかを以下に説明します．

1）システムLSIなどの組み込みCPUとのアクセス

CPUとIPコアが直接つながっている場合，アセンブリ言語やC言語など適当な言語を用い，IPコアのアドレスを直接指定して読み書きを行います．例えばC言語の場合は，アドレス値を入れたポインタpを用意して次のように記述すれば読み書きできます．

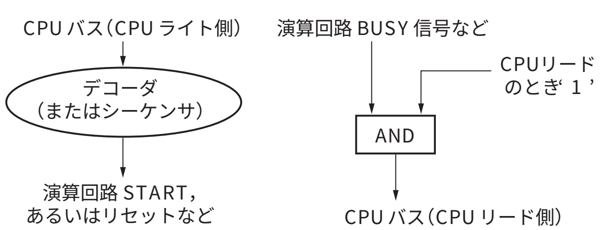
```
volatile int  *p = アドレス;
value = *p;      // 指定アドレスのリード
*p = value;      // 指定アドレスへライト
```

ここで，十分に注意すべき点があるので，以下にまとめておきます（C言語に限らず，ほかの言語でも同じ）．

- **ワード境界**――ターゲットCPUとIPコアにおいて何バイトを1アドレスとするか（例えば，バイト・アドレッシングやワード・アドレッシング），あるいは複数バイト（例えば32ビット）のデータ・バスに対してバイト単位でのリード/ライトを行えるかどうか，という点に注意する必要がある．もしこれらがCPUとIPコアで食い違っているのであれば，ポインタの型を含め，CPUプログラミングに相当気を配らないと，CPUとコアでうまくデータをやり取りできない．
- **エンディアン**：1ワードが複数バイトからなるとき，リトル・エンディアンかビッグ・エンディアンかは，ターゲットとなるCPUによって異なる．CPUとIPコアでエンディアンが異なる場合は，CPUプログラムのほうでエンディアンを変換しなければならない．
- **volatile（揮発性）修飾子**――ポインタの宣言にvolatile修飾子を忘れずに付ける必要がある．これは，「そのポインタの指すアドレスを読んだとき，前回読んだ値と同じとは限らないし，一度書いた値がそのまま保持されるとは限らない」ことを表す修飾子．IPコア内部のレジスタ（とくに制御レジスタ）は，CPUが読み書きしなくてもIPコアのほうで値を変えてしまうことがあるので，それをコンパイラに教えるためにこれを付ける．付け忘れると，コンパイラのコード最適化によってプログラムがうまく動作しなくなる可能性がある．

また，IPコアを駆動するための典型的なプログラム例を**リスト1**に示します．**図4**に示したシーケンス・チャートに従って，コアのレジスタを読み書きしているだけです．**リスト1**ではポーリングによってIPコアの動作終了待ちを行っていますが，割り込みを使う場合には，演算スタートをかける前に割り込みベクタ設定や割り込み許可などの処理を行っておきます．詳しくは，ターゲットCPUのコン

4




```
assign 演算回路START
       = (レジスタ・アドレス == 制御レジスタのアドレス
       && CPUライト == 1'b1
       && CPUライト・データ[該当ビット] == 1'b1)
       ? 1'b1 : 1'b0;
assign CPUリード・データ
       = (レジスタ・アドレス == 制御レジスタのアドレス
       && CPUリード == 1'b1) ? {0, 演算回路BUSY} : 0;
```

図8 典型的な制御レジスタ回路とVerilog HDL記述例

制御レジスタは，CPUからの書き込み時に必要なパルスを生成したり，CPUからの読み出し時に演算回路の状態を返したりするためのデコーダ（またはシーケンサ）によって実現される．かならずしもDフリップフロップ（D-FF）があるわけではない．

リスト1 IPコア制御のプログラミング例（C言語）

図3のレジスタを想定し，ポーリングを使用した場合のプログラミング例を示す．図4のシーケンス・チャートどおりにIPコアのレジスタを読み書きしていけばよい．もしIPコアがすぐBUSYを立てない場合は，いったんBUSYが立つまで待つこと．

```
while (ReadIPcore(制御レジスタ・アドレス) != 0)
       ;                                          // コアがBUSYでなくなるまで待つ
WriteIPcore(データレジスタ・アドレス, 入力データ);
WriteIPcore(制御レジスタ・アドレス, 2);           // 演算スタート
                                                     (ここでは，ビット1に1を書くとスタート)
while (ReadIPcore(制御レジスタ・アドレス) != 0)
       ;                                          // 演算中の間は待つ（ポーリング）
result = ReadIPcore(データレジスタ・アドレス);    // 結果取り出し
```

注：指定アドレスの読み書きを行う関数ReadIPcore，WriteIPcoreを別途作っておく（読み書きの方法は本文を参照）．

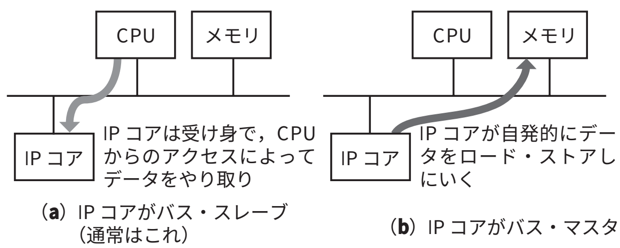


図9　マスタとスレーブ

通常，IP コアへデータの出し入れを行うのは CPU であり，IP コアは CPU から命令を受けるまでは受け身で待っている（スレーブ・モード）．しかし，IP コアが自発的にデータを取りにいったり書き込んだりする場合もある（マスタ・モード）．このような場合，実装は複雑になるものの，CPU の負荷を軽減したり，バスの占有率を低減するなどして，データ転送の高速化を図ることができる．

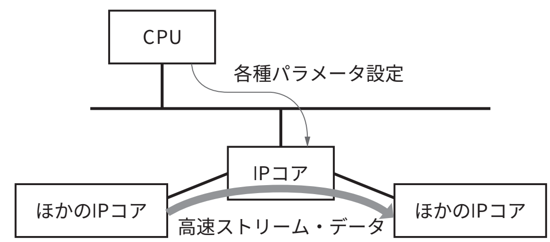


図10　別の IP コアとの直接接続

IP コアの話し相手は CPU だけとは限らない．とくに，高速ストリーム・データを処理する場合などは IP コアへ連続的にデータを投入する必要があるので，いちいち CPU を介してデータを転送するのではなく，ほかの IP コアとの間に直接バスを張って受け渡しする．CPU からは各種パラメータ設定などを行う．

パイラのマニュアルなどを参照してください．

2）パソコンなどから通信経路を介してアクセス

IP コアが CPU と間接的につながっており，CPU のアドレス空間にマップされない場合もよくあります．例えば，IP コアをアクセラレータ・ボードなどに載せてパソコンに挿す場合や，IP コアを FPGA ボードに載せてテストする場合などです．

このような場合，PCI バスや USB，パラレル・ポート，シリアル・ポート（RS-232-C）などの通信経路を介して，パソコンなどから IP コアのレジスタにアクセスすることになります．このとき，いろいろなハードウェア/ソフトウェアの階層を作ってデータを転送していくのですが，もっとも注意が必要なのは階層間でデータ転送を確実に行うことです[注6]．例えば，パソコンと IP コアはそれぞれが異なる速度でばらばらに（並列に）動いていますが，両者が歩調を合わせてデータを交換できるようなしかけを作る必要があります（具体的な例は後述）．

● 実用化のための接続方法の基礎

IP コアのレジスタ・インターフェースの典型的な設計法を説明してきましたが，実用回路ではいろいろとアレンジする場合があります．代表的なものを以下に紹介します．

1）バス・マスタとバス・スレーブ

これまでに説明した IP コアの使いかたは，IP コアは CPU からデータを入出力されるのを受け身で待っている，という形態です（**図9（a）**）．このような IP コアはバス・スレーブと呼ばれるもので，簡単なのでもっともよく使われる方式です．しかし，データ転送を「メモリ→CPU→IP コア」というようにいちいち CPU を介して行うため，バスの占有率が上がったり，CPU に負荷がかかるという問題があります．

このため，高速に大量のデータを扱う場合は，コアのほうが自発的にメモリからデータをロード・ストアするほうがベターです（**図9（b）**）．つまり，コアが自発的にバスの使用権を取りにいってデータ転送を行います．このような IP コアをバス・マスタと呼びます．

2）コアのデータ経路を CPU バスにつながない方法

さらに高速で連続的なデータの転送が必要な場合，IP コアのデータ入出力を CPU バスに接続せず，ほかの IP コアなどと直結してしまうやりかたもあります（**図10**）．CPU は，IP コアの動作モード設定やリセット，起動などのコマンドを発行する役割を担います．この構成では，データは IP コアのレジスタ・インターフェースを通らず，演算回路へ直接出入りすることになります．

3）レジスタ・インターフェースと演算回路の間のやり取り

これまで説明したように，IP コア内部ではレジスタ・インターフェースと演算回路の間でデータ転送を行います．そのやりかたについては，先述したようにパラレルでデータを全ビット一度に渡すというケースだけでなく，バス幅を減らすために少ビットずつシーケンシャルに送るケースもあります．また，コアとレジスタ・インターフェースでクロックが異なる場合もあります（こうした違いを CPU から隠ぺいするのがレジスタ・インターフェースの役割）．

このようなさまざまなデータ転送プロトコルや非同期系とのインターフェースについては，参考文献（1）を参照してください．

注6：IP コアとパソコンの場合に限らず，プログラミングや通信などでも，並列に動作する二つの物（プロセス）の間でデータのやり取りを確実に行うのは，かなりの注意を必要とする．

3 インターフェースの作成例

これまでに説明した内容を，実際に回路やソフトウェアを組んで実験してみました．理解しやすくするため，ここに挙げる例題はかなり単純化しています（ここで紹介するデータは本誌ホームページhttp://www.cqpub.co.jp/dwm/からダウンロードできる）．

● ポイントは「回路の差し替えが容易な構成」

ここでは，「乗算回路をFPGA（またはCPLD）に実装し，これをRS-232-C経由でパソコンのソフトウェアから動かす」という実験を行います．この例題のポイントは，冒頭でも述べたとおり，なるべく回路の差し替えが容易になるように設計することです．つまり，乗算回路を割り算回路にしたり，データ転送をRS-232-CではなくPCIバス経由にしたり，といった変更が簡単に行える構成にします．

FPGAとパソコンを合わせた全体構成を**図11**に示します．パソコンからFPGAに向かって，いくつかの階層を切ってデータの受け渡しを行います．ここでの階層の切りかたは，わりと典型的な形にしてあります．以下，**図11**のパソコン側から順に概要を説明します．

アプリケーション・プログラムは，乗算処理を使うプログラムです．ここから，一つ下の階層であるIPコア駆動API（application program interface）の乗算実行関数を呼び出すことで，回路に乗算を行わせることができます．IPコア駆動APIは，その下のCOM1デバイス・ドライバを使って，IPコアのレジスタ読み書きを行います．COM1デバイス・ドライバは，シリアル・ポートを使ってバイト単位でデータの送受信を行います．このとき，IPコアにアクセスするためにどのようなバイト系列をパソコンとFPGAの間でやりとりするか，というデータ転送のしかた（プロトコル）を適宜決めておきます．ここまでがパソコン上のソフトウェアの処理で，以下はFPGA側になります．

まず，RS-232-Cインターフェース回路がバイト単位のデータを受け取ります．次のコマンド解釈回路は，受け取ったバイト系列を解釈して，IPコアのレジスタをどう読み書きすればよいかを割り出します．最後に，乗算IPコアにデータが渡って演算が行われます．

このように階層を切っておけば，例えばRS-232-CをPCIバスに変更する場合，パソコン側のデバイス・ドライバとFPGA側のRS-232-Cコアという対になっている部分だけを入れ替えればよく，そのほかのハードウェアやソフトウェアを変更する必要はありません．

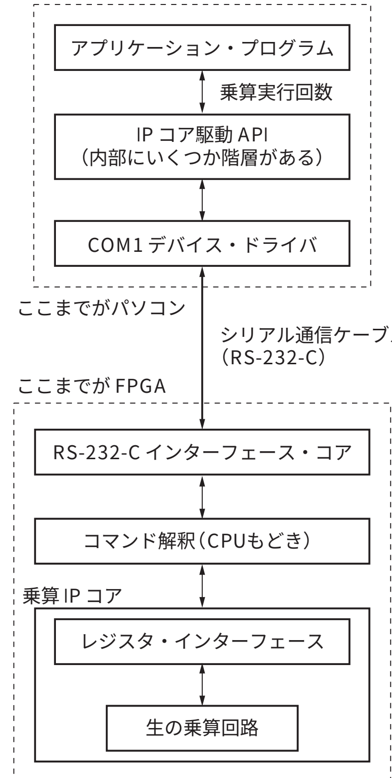


図11　例題回路（乗算回路をパソコンから動かす）
パソコン内部のソフトウェアおよびFPGA内部の回路にさまざまな階層を作り，階層間でデータを受け渡ししていく．パソコンの最上位ソフトウェアから見ると，ただ演算関数を呼ぶだけでIPコアを駆動でき，RS-232-Cなどの途中階層の動作は隠ぺいされる．このように階層を切ると，例えばRS-232-CをPCIバスにするような変更が容易になるというメリットがある．

● FPGA内部はバス接続型にして変更を容易に

FPGA内部の回路構成を**図12**に示します．**図11**のとおりに回路を直列につないでもよいのですが，今回はシステムLSIなどと同じく，IPコアと全体動作コントローラ（CPUに相当）がバスでつながった構成をとることにしました．データ・バス幅，アドレス・バス幅はそれぞれ8ビットです．なお，**図12**の全体動作コントローラが，**図11**のコマンド解釈部に相当します．このコントローラは，RS-232-Cインターフェース・コアから受け取ったバイト系列を解釈し，それに基づいて乗算IPコアを駆動します．

このようなバス接続型の回路構成をとることで，乗算以外の演算コアを追加したり，全体の動作シーケンスを変更

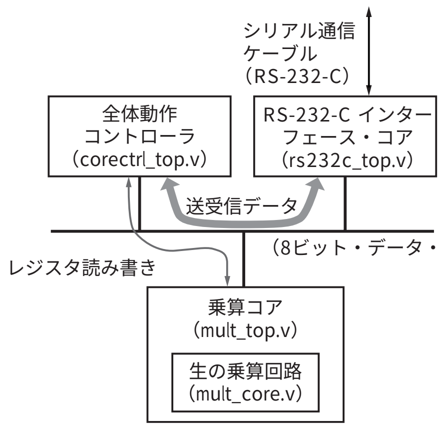


図12　FPGA内部回路構成

今回はシステムLSIを模擬して，IPコアと全体動作コントローラ（CPUに相当）がバスにつながった形態をとった．つねにこのような構成をとる必要はないが，演算回路の増設やインターフェースの変更，制御シーケンスの変更などを簡単に行える，というメリットがある．今回はIPコアは全部バス・スレーブである．

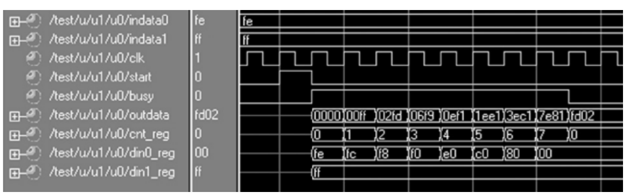


図13　乗算回路の動作タイミング・チャート

indata0/1に演算入力をセットしてSTARTパルス（start）を与えると，9クロック後に演算出力が得られる．

しようとしたとき，回路全体を作り直さなくてすみます．

それでは，ソフトウェアやハードウェアの詳細について，今度はFPGA側からパソコン側へ（**図11**の下から上へ）向かって見ていきましょう．まず，乗算回路本体のVerilog HDL記述（mult_core.v）と動作タイミング・チャートを，それぞれ**リスト2**，**図13**に示します．この回路は，8ビットの二つの入力の積を筆算と同様の方法で計算するものです．とくに高速化のくふうを施しているわけではないので，演算方法についての説明は割愛します．この回路の動作タイミングは**図5（c）**と同じ典型的なもので，STARTパルスによって起動し，演算中にBUSY信号を出力します．

次に，乗算回路本体にレジスタ・インターフェースをラッパとしてかぶせます．

そのためにまず，どのようなレジスタを設けるかを検討し，**表1**のようなレジスタ・マップを作っておきます．今回の乗算IPコアのレジスタ構成は，**図3**で説明した基本構成にのっとっており，データを入出力するためのデータ・レジスタと，コアを駆動するための制御レジスタからなり

リスト2　乗算回路本体（mult_core.v）

```
module mult_core (indata0, indata1, outdata, start,
                                            busy, clk, rstn);
    input   [7:0]   indata0;          // 乗算引き数その1
    input   [7:0]   indata1;          // 乗算引き数その2
    output  [15:0]  outdata;          // 乗算結果
    input           start;            // 演算STARTパルス
    output          busy;             // BUSY（演算中）
    input           clk;              // クロック
    input           rstn;             // リセット（負論理）

    reg [7:0]   din0_reg;             // 引き数1レジスタ
    reg [7:0]   din1_reg;             // 引き数2レジスタ
    reg [15:0]  dout_reg;             // 結果レジスタ
    reg [2:0]   cnt_reg;              // ループ・カウンタ
    reg         busy_reg;             // BUSYフラグ

    always @(posedge clk or negedge rstn) begin
        if (rstn == 1'b0) begin
            busy_reg    <= 1'b0;
        end else if (busy_reg == 1'b0 && start
                                            == 1'b1) begin
      // STARTパルスが来たら各レジスタを初期化
            din0_reg    <= indata0;
            din1_reg    <= indata1;
            dout_reg    <= 16'd0;
            cnt_reg     <= 3'd0;
            busy_reg    <= 1'b1;
        end else if (busy_reg == 1'b1) begin
      // 結果レジスタをシフトし，乗数を加算
            if (din0_reg[7] == 1'b1) begin
                                        //  （シフト＆加算）
                dout_reg    <= {dout_reg[14:0],
                                1'b0 + {8'd0, din1_reg};
            end else begin          //  （加算せずシフトのみ）
                dout_reg    <= {dout_reg[14:0], 1'b0};
            end
            din0_reg    <= {din0_reg[6:0], 1'b0};

      // ループ回数カウント(8回まわったら終わり)
            cnt_reg     <= cnt_reg + 3'd1;
            if (cnt_reg == 3'd7) begin
                busy_reg    <= 1'b0;
            end
        end
    end

    assign  outdata = dout_reg;
    assign  busy    = busy_reg;
endmodule
```

ます．

また，CPU側から見たレジスタ・インターフェースの動作タイミングも検討します．ここでは，**図5（a）**，**（b）**に示したシンプルなものにしてあります．**図14**に，シミュレーションのようすを示します．

以上の検討結果を踏まえて作成したレジスタ・インターフェース回路のHDL記述を**リスト3**に示します．ここでは，レジスタ・インターフェース内部に生の乗算回路を組み込む形にしたため，ポートはCPUとの接続信号のみになっています．CPUからの信号をデコードし，データ・レジスタへの書き込みや乗算回路のスタート，読み出しデータの出力などを行います．これらの動作を行うための基本回路は，**図6**～**図8**に示したとおりです．

表1　IP コアのレジスタ・マップ

今回は，乗算コア（上半分）とRS-232-C通信コア（下半分）を使うことにした．表中，影が付いている部分が各IPコアの制御レジスタ，それ以外がデータ・レジスタである．

	アドレス（8ビット）	レジスタ名	R/W	説　明
乗算コア	00h	乗算トリガ	W	ビット0：演算START
	(00h)	乗算BUSY	R	ビット0：演算中
	01h	乗算入力0	W	演算入力（8ビット）
	02h	乗算入力1	W	演算入力（8ビット）
	03h	乗算出力0	R	演算結果（下位8ビット）
	04h	乗算出力1	R	演算結果（上位8ビット）
RS-232-C通信コア	10h	送受信状態	R	ビット0：1バイト送信中 ビット1：1バイト受信済み
	11h	受信データ	R	受信データ（8ビット） ここを読むと，受信済みフラグは自動で降りる
	12h	送信データ	W	送信データ（8ビット） ここにデータを書くと，自動で送信開始

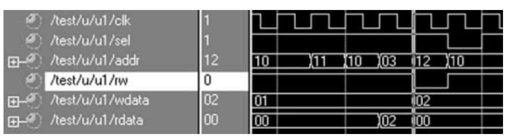


図14　乗算IP レジスタ・インターフェースの動作タイミング・チャート例

普通のSRAMと同様に，アドレスaddr，セレクト信号sel，リード/ライト信号rwを与えてアクセスする．バス・サイクルは1クロック．この図では，アドレス12hにデータ02hを書こうとしているところなどが示されている．

リスト3　乗算IP コアのレジスタ・インターフェース（mult_top.v，抜粋）

```
module mult_top (addr, wdata, rdata, rw, sel,
                                              clk, rstn);
    input   [7:0]   addr;           // レジスタ・アドレス
    input   [7:0]   wdata;          // ライト・データ
    output  [7:0]   rdata;          // リード・データ
    input           rw;             // リード時1，ライト時0
    input           sel;            // レジスタ・アクセス時1
    input           clk;            // クロック
    input           rstn;           // リセット（負論理）

    parameter   ADR_MLT_TRIG  = 8'h00;
                                    // レジスタ・アドレス
    …（ほかは省略）…

    reg [7:0]   din0_reg;           // データ・レジスタ
    reg [7:0]   din1_reg;           // データ・レジスタ
    wire [15:0] dout_node;          // 乗算出力

   // レジスタのリード/ライト信号のデコード
    assign  write_trig  = (addr == ADR_MLT_TRIG  && sel
                == 1'b1 && rw == 1'b0) ? 1'b1 : 1'b0;
                          // トリガ・レジスタ書き込み時1
   …（ほかは省略）…

   // 演算回路STARTパルスのデコード
    assign  start_node  = write_trig & wdata[0];

    mult_core u0 (                  // 乗算回路本体とつなぐ
        .indata0(din0_reg),         // 演算入力その1
        .indata1(din1_reg),         // 演算入力その2
        .outdata(dout_node),        // 演算結果
        .start(start_node),         // STARTパルス
        .busy(busy_node),           // BUSY信号
        .clk(clk),
        .rstn(rstn)
    );

    always @(posedge clk) begin// データ・レジスタ本体
        if (write_din0 == 1'b1) begin
            din0_reg    <= wdata;// データ・レジスタ書き込み
        end
        …（ほかは省略）…
    end

  // コアへのリード・アクセスがあったとき，アドレスに応じて
  // BUSYステータスや演算結果などをバスへ出力する．
    assign  rdata = (read_busy == 1'b1) ?
                                        {7'd0, busy_node} :
                (read_dout0 == 1'b1) ?
                                            dout_node[7:0]  :
                (read_dout1 == 1'b1) ?
                                           dout_node[15:8]  :
                  8'd0; //（アクセスされていなければ0を出力）
endmodule
```

● 全体動作のコントロール

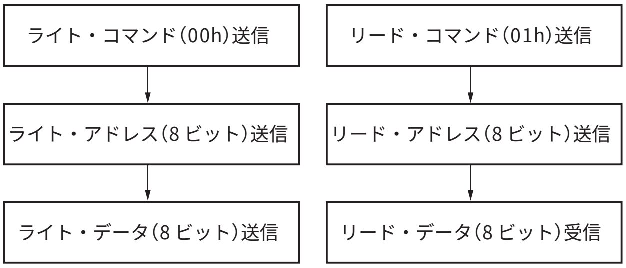
RS-232-C通信コアのレジスタについては，**表1**を参照してください．乗算コアとあまり大きな違いはありません．制御レジスタによって送受信の状況をチェックし，データ・レジスタによって送受信データをやり取りします．RS-232-C通信コア本体の組みかたについては，誌面のつごう上ここでは割愛します（今回は参考文献(3)とほぼ同じ回路を使ったので，そちらを参照していただきたい）．

さて，パソコンから乗算コアのレジスタを読み書きするには，レジスタのアドレス値やデータ値をFPGAとパソコンの間で受け渡しできるようにする必要があります．ここで，RS-232-Cでは一度の交信でたかだか8ビットの値しか受け渡しできないので，複数回の交信がどうしても必要になります．

そこで今回は，データのやり取りの方法を**図15**のように決め，1回のレジスタ・アクセスを3回のRS-232-C交信によって行うことにしました．もちろん，アドレスやデータのビット長が異なる場合は，交信のしかたが変わります．

図12の全体動作コントローラは，乗算コアやRS-232-Cコアのレジスタを読み書きしつつ，**図15**のとおりに通信を行う回路です．今回は，**図16**のようなステート・マシンを組みましたが，もっと複雑な動きが要求される場合にはCPUを使うほうがよいでしょう．具体的なHDL記述(corectrl_top.v)は，ごく普通にステート・マシンを書いてあるだけなので，ここでは割愛します．

最後に，**図12**のとおりに乗算コア，RS-232-Cコア，全体動作コントローラを接続してFPGAのトップ階層ができ上がります（本誌ホームページからダウンロードできるファイルfpga_top_all.vを参照のこと）．FPGA全体では，RS-



（**a**）IPコアのレジスタへのライト　（**b**）IPコアのレジスタのリード

図15　RS-232-Cを介したデータ転送プロトコル例（送受信はパソコンから見た場合）

通信経路に応じて，データ転送プロトコルを決めておく必要がある．今回は図のように，パソコンからRS-232-C経由でIPコアのレジスタにアクセスできるようにプロトコルを決めた．プロトコルは，通信経路で一度に送ることのできるデータのビット数やデータの内容に依存して大きく変わる．ここに示したのはあくまでも一例．

232-CのTxD，RxD，CTS，RTS信号，クロック，およびリセットだけがI/Oピンとして出る形になります．

● ソフトの階層構成はハードと逆の順番になる

以上のようにして作ったFPGAを動かすため，パソコン側で専用のソフトウェアを準備する必要があります．ここでは，それを「コア駆動API」と呼びます．ここで，もっとも注目していただきたいのは，

- ソフトウェア側も階層構成をとる
- その階層はハードウェア側と逆の順番になる

ということです．

図17に，今回作成したAPIの階層構成を示します．ハードウェアでは，「RS-232-Cのシリアル・データ通信→**図15**のバイト転送→**図4**のレジスタ読み書き（→乗算）」という順にデータ転送プロトコルを変換しましたが，APIではちょうどこの逆の順番にプロトコルを変換します．つまり，プロトコルの逆変換を行うのがAPIであり，その階層構成はハードウェアと対称になります．

それでは，さっそく実例を見てみましょう．ハードウェアに近い側から上へ向かって説明します．

- **レイヤ0** ――レイヤ0は，バイト単位でRS-232-C転送を行うための関数群を提供する（**リスト4**）．この部分はOSに依存するが，ここではWindowsにおける例を示す．RS-232-Cのみならず，ハードウェアとのデータ転送は，ファイルへの読み書きと同じように扱うことが多い．具体的には次のようにする．最初に，CreateFile( )というWin32 API[注7]を使ってRS-232-Cをオープン（初期化）する．通信パラメータもこのとき設定しておく．データの読み書きはWriteFile( )，ReadFile( )というAPI関数で行う．処理が終わったら，CloseHandle( )でクローズする．
- **レイヤ1** ――レイヤ1は，IPコアのレジスタ読み書き関数群を提供する（**リスト5**）．**図15**に記したプロトコルに従って，RS-232-Cとデータ送受信を行う．FPGA側の「全体動作コントローラ」とは逆の動作を行うわけである．この階層はOSに依存しないが，通信経路には依存する．つまり，RS-232-CをPCIなどのほかのインターフェースに変更した場合，この階層の修正が必要になる．
- **レイヤ2** ――レイヤ2は，乗算回路を動かして演算を行う関数を提供する（**リスト6**）．**リスト1**と同様に，乗算回路のレジスタを読み書きすることで演算を行う．この

注7：windows.hをインクルードする必要がある．

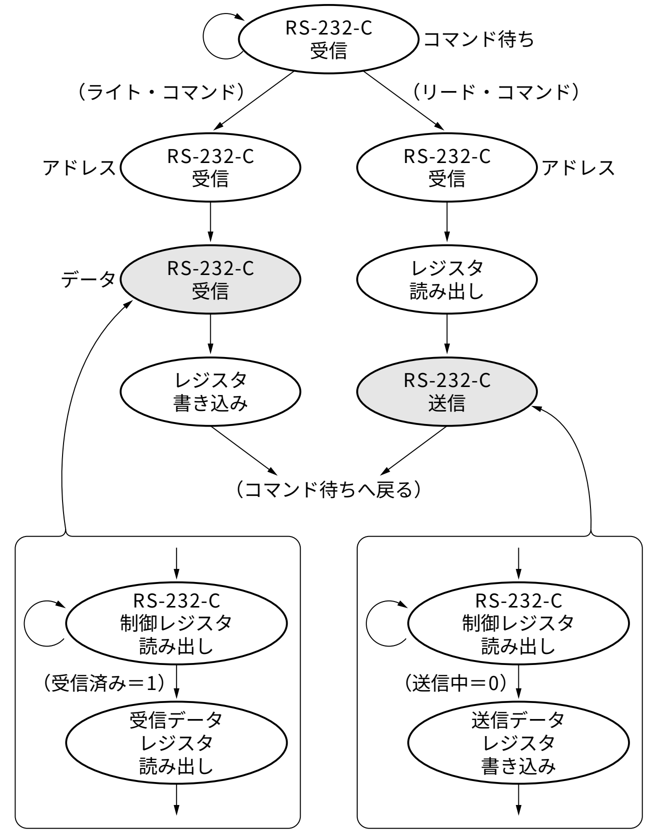


図16　全体動作コントローラの状態遷移図

FPGAの全体動作コントローラは，図15のプロトコルに基づいて組み立てられた状態遷移図に従って動作する．今回はステート・マシンで回路を組んだが，もちろんCPUを使ってもよい．なお，RS-232-Cの送受信は，RS-232-Cコアの制御レジスタやデータ・レジスタを図の下段のとおりに読み書きすることで行う．

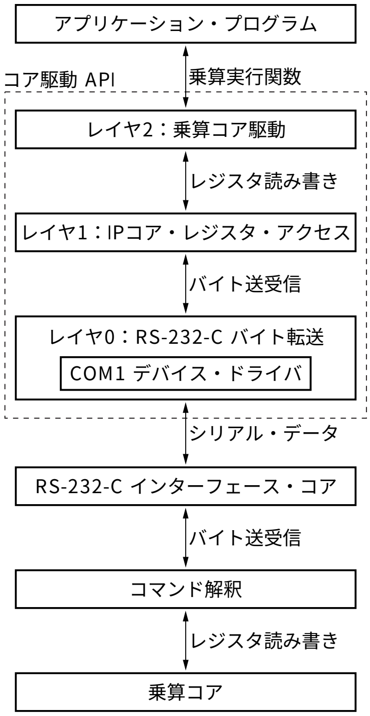


図17　IPコア駆動API（ソフトウェア）のレイヤ構造

パソコン側のソフトウェアも階層構造をとるようにした．この場合のもっとも大きな特徴は，ハードウェア側とは逆の順番にプロトコル変換を行うことである．アプリケーションへは「乗算実行」のコマンドをAPIとして提供し，下の階層へ向かってレジスタ読み書き，バイト送受信…というようにデータ転送プロトコル変換を行う．結果として，レイヤ0とRS-232-Cインターフェース・コア，レイヤ1とコマンド解釈，レイヤ2と乗算コアの動作が対応することになる．

リスト4　パソコン側のプログラム（mult_driver.c，抜粋）

API レイヤ0（最下位レイヤ）のRS-232-C通信部のプログラムを示す．

```
#include <windows.h>     // これを忘れずに付ける
#include <stdio.h>

// RS-232-Cをオープン，初期化．ファイル・ハンドラを取得
BOOL OpenRS232C(LPCTSTR fname, HANDLE *handler, DWORD BaudRate, BYTE
Parity, BYTE ByteSize, BYTE StopBits)
{
    *handler   = CreateFile(fname,
                    GENERIC_WRITE | GENERIC_READ,
                    0, NULL, OPEN_EXISTING,
                    FILE_ATTRIBUTE_NORMAL,  NULL);
    if  (*handler == INVALID_HANDLE_VALUE) {   // error
        *handler     = NULL;
        fprintf(stderr, “ERROR: CreateFile()¥n”);
        return (FALSE);
    }
    else {  // success: configure RS-232-C parameters
        DCB dcb;
        SetupComm(*handler, 1024, 1024);  // 通信バッファ確保
        PurgeComm(*handler, PURGE_TXABORT | PURGE_RXABORT |
            PURGE_TXCLEAR | PURGE_RXCLEAR);   // 通信ステータス初期化
        // 以下，RS-232-Cパラメータ設定
        GetCommState(*handler, &dcb);
        dcb.BaudRate    = BaudRate;        // ボーレート
        dcb.ByteSize    = ByteSize;        // データ・ビット数
        dcb.Parity       = Parity;         // パリティ長
        dcb.StopBits     = StopBits;       // ストップ・ビット数
        if (SetCommState(*handler, &dcb) == FALSE) {
            fprintf(stderr, “FAILURE: SetCommState()¥n”);
            return (FALSE);
        }
        GetCommState(*handler, &dcb);
        return (TRUE);
    }
}
// RS-232-Cをクローズ
void CloseRS232C(HANDLE handler)
{
    if((handler) != NULL)
        CloseHandle(handler);
}
// 1バイト送信
BOOL SendRS232C(HANDLE handler, LPCSTR WriteData)
{
    DWORD tmp;
    return (WriteFile(handler, WriteData, (long)1, &tmp, NULL));
}
// 1バイト受信
BOOL ReceiveRS232C(HANDLE handler, LPSTR ReadData, DWORD *Bytes)
{
    return (ReadFile(handler, (LPVOID)ReadData, (long)1, Bytes, NULL));
}
```

```
// IPコアのレジスタへデータを書き込む
void    WriteIPcore(HANDLE handler,
             char address, char data)
{
    char    sndbuf;
    sndbuf  = (char)0x00;
    SendRS232C(handler, (LPSTR)&sndbuf); //コマンド送信
    sndbuf  = address;
    SendRS232C(handler, (LPSTR)&sndbuf); // アドレス送信
    sndbuf  = data;
    SendRS232C(handler, (LPSTR)&sndbuf); // データ送信
}

// IPコアのレジスタを読み出す
char    ReadIPcore(HANDLE handler, char address, char *data)
{
    char    sndbuf, rcvbuf;
    DWORD   rcvbyte;
    sndbuf  = (char)0x01;
    SendRS232C(handler, (LPSTR)&sndbuf); // コマンド送信
    sndbuf  = address；
    SendRS232C(handler, (LPSTR)&sndbuf); // アドレス送信
    while (ReceiveRS232C(handler, (LPSTR)&rcvbuf, &rcvbyte) == FALSE)
        ;                                 // リード・データを受信するまで待つ

    *data   = rcvbuf;
    return (rcvbuf & 0xFF);
}
```

◀リスト5
パソコン側のプログラム（mult_driver.c，抜粋）

API レイヤ1のIPコア・レジスタ・アクセス部のプログラムを示す．

4

リスト6
パソコン側のプログラム(mult_driver.c，抜粋)
API レイヤ2(最上位レイヤ)の乗算コア駆動部のプログラムを示す．

```
// 8ビット値din0×din1の値を計算し，結果の上位8ビットを*dout1，下位8ビットを*dout0にストアする
void    RunMULTcore(HANDLE handler, char din0, char din1, char *dout0, char *dout1)
{
    char    rdbuf;
    while (ReadIPcore(handler, 0, &rdbuf) != 0) // コアがBUSYでなくなるまで待つ
        ;
    WriteIPcore(handler, 1, din0);              // 乗算引き数1セット
    WriteIPcore(handler, 2, din1);              // 乗算引き数2セット
    WriteIPcore(handler, 0, 1);                 // 演算START
    while (ReadIPcore(handler, 0, &rdbuf) != 0) // 演算終了まで待つ
        ;
    ReadIPcore(handler, 3, dout0);              // 演算結果下位読み出し
    ReadIPcore(handler, 4, dout1);              // 演算結果上位読み出し
```

▼リスト7　パソコン側のプログラム(mult_driver.c，抜粋)
アプリケーション部のプログラムとその実行結果を示す．

```
void main(void)
{
    HANDLE  handler;                        // COM1デバイスのファイル・ハンドラ
    char    din0, din1, dout0, dout1;       // 乗算コアとの入出力用変数

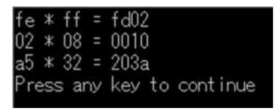
    if (OpenRS232C(“COM1”, &handler, (DWORD)9600, (BYTE)0, (BYTE)8, (BYTE)0) == TRUE) {
    // 9600bps，パリティなし，8ビット・データ，ストップ・ビット1でRS-232-Cをオープン

    // 乗算1回目
        din0 = (char)0xFE; din1 = (char)0xFF;                 // 乗算の入力値をセット
        RunMULTcore(handler, din0, din1, &dout0, &dout1);     // 乗算コアをキック
        fprintf(stderr, “%02x * %02x = %02x%02x¥n”, din0 & 0xFF, din1 & 0xFF, dout1 & 0xFF, dout0 & 0xFF);

    // 乗算2回目
        din0 = (char)0x02; din1 = (char)0x08;
        RunMULTcore(handler, din0, din1, &dout0, &dout1);
        fprintf(stderr, “%02x * %02x = %02x%02x¥n”, din0 & 0xFF, din1 & 0xFF, dout1 & 0xFF, dout0 & 0xFF);

    // 乗算3回目
        din0 = (char)0xA5; din1 = (char)0x32;
        RunMULTcore(handler, din0, din1, &dout0, &dout1);
        fprintf(stderr, “%02x * %02x = %02x%02x¥n”, din0 & 0xFF, din1 & 0xFF, dout1 & 0xFF, dout0 & 0xFF);
    }

    CloseRS232C(handler);           // RS-232-Cをクローズ
}
```



階層はOSにも通信経路にも依存しないが，IPコアに依存する．つまり，今回の乗算回路と別のIPコアをつなげた場合には，それに応じて関数を作る必要がある．

- **アプリケーション**――アプリケーションは，乗算を利用する部分である(**リスト7**)．このレベルから見たとき，実際に乗算回路を動かして結果を得ているのか，あるいはソフトウェアで乗算を実行しているのかは，区別がつかない．

4 例題の回路の動作を実機で確認する

上記の例題の動作を，次の三つのFPGAボードを使って動作確認しました．

- 本誌2005年1月号付属FPGA基板
  (米国Xilinx社のSpartan-3「XC3S50」搭載)[6]
- Stratix評価ボード
  (CQ出版製．米国Altera社のStratix「EP1S10」搭載)[5]
- CPLDボード(ソリトンウェーブ製．Xilinx社のCoolRunner-II「XC2C256」を搭載)[4]

このほかにも，RS-232-Cインターフェース(シリアル・インターフェース)を備えたCPLD/FPGAボードであれば，ほとんどのもので動作するはずです．Spartan-3の場合で160Sliceと小規模な回路なので，大半のFPGAはもちろん，CPLDにも入ります．

なお，実験にあたっては，ボード上のRS-232-C，クロック，リセットの各信号がCPLD/FPGAのどのピンに来ているかを調べ，適宜ピン割り当てを行って論理合成してください．また，クロック周波数によってはRS-232-Cコアの設定を変更する必要があります(詳細は，本誌ホームページのダウンロード・ページの本稿関連データ内にあるREADME.txtを参照)．

Windows用ソフトウェアについては，**リスト4～リスト7**をmult_driver.cという1本のCソースにまとめたものを用意してあるので，これをVisual C++などで「Win32

Console Application」としてコンパイルしてください．

CPLD/FPGAボードとパソコンをシリアル・ケーブルでつないだ後，上記のソフトウェアを走らせると，**リスト7**に示したように乗算結果が表示されます．

● **実用時の重要な三つの注意点**

今回用いた例題はかなりシンプルなものですが，基礎的な知識はほぼ網羅しています．ただし，実際に用いるうえではいくつか注意するべき点があるので，以下にそれを説明します．

1）同期ずれなどによるデッドロック

通信エラーやそのほかの原因によってCPLD/FPGAとパソコンの動きがかみ合わなくなると（例えば，一方がRS-232-Cの送信待ちなのに他方が受信待ちに入ったりすると），処理がおかしくなったり，永久停止（デッドロック）したりしてしまいます．両者の歩調がずれてしまったわけですが，これを「同期ずれ」と言います．

最低限の対策として，両者をリセットしてから動かし始めるようにする必要があります[注8]．もし可能なら，同期ずれが起こったことを自動検知して同期を回復するようにプロトコルを組んだり，タイムアウトによってデッドロックを検出してセルフ・リセットするような方法が有効です．ただし，どのような同期ずれからでも自動回復できるようにすることは，それほど簡単なことではありません．

2）動作速度の違いによるデータ転送ミス

今回は**図15**などで，パソコンから連続してデータを送信していますが，もしCPLD/FPGA側が非常に低速であったとすると，データを受け取りきれずに取りこぼしてしまうようなケースも起こりえます．

また，**リスト1**や**リスト4**でIPコアをスタートさせたとき，IPコアがBUSY信号を立てる前にパソコンがBUSY信号を読んでしまい，パソコンが「すでに演算が終わった」と勘違いする場合もあります（俗に「BUSY待ちをすっぽ抜ける」と言う）．

いったんこのような現象が発生すると，上記の同期ずれが起きて誤動作してしまいます．パソコンとFPGAのように並列に動作するものどうしの間でデータをやり取りするときには，かなり頻繁にぶつかる問題です．

もっとも抜本的な解決は，何かのデータを一つ送ったら，かならず相手からアクノリッジを送り返してもらい，それから次のデータを送ることです[(1)]．つまり，連続データ転送を行わず，毎回ハンドシェイクを実施するわけです．ただし，あまり厳密に適用すると通信速度が劇的に落ちてしまうので，ケース・バイ・ケースで検討してください．

3）非同期クロック系との接続による回路誤動作

有名な話ですが，回路の入力にクロック系統の違う信号をつなぐと，回路が誤動作する場合があります．チップの外から来る信号も同様です．シンクロナイザ（同期化回路）を入れるなどの対策がありますが，詳しくは参考文献(1)を参照してください．

＊　＊　＊

本稿ではさまざまな事項を紹介しましたが，細かいことはともかく，次の二つがもっとも重要です．

- ハードウェアにも，それを駆動するソフトウェアにも，階層を作る
- ソフトウェアからハードウェアを駆動しようとするなら，両者は並列に動くので，並列プロセス・プログラミングのセンスで考える

注8：どちらを先にリセット解除するか，一方のリセット解除からどれだけあとに他方を解除するか，ということにも気をつける必要がある．

参考・引用*文献

(1) 森岡澄夫；HDLによる高性能ディジタル回路設計，CQ出版，2002年11月．

(2) 下馬場朋禄；ディジタル信号処理アルゴリズムをFPGAに実装する，Design Wave Magazine，pp.77-103，2004年11月号．

(3) 森岡澄夫；受信しながら送信する並列処理の回路を作る，トランジスタ技術，pp.186-192，2003年5月号（RS-232-C送受信回路の説明）．

(4) ソリトンウェーブのホームページ，http://www.solitonwave.co.jp/（CPLDボード「HDLトレーナー」の説明．トランジスタ技術2003年5月号でも紹介）．

(5) Stratix評価キットに関するホームページ，http://www.cqpub.co.jp/eda/Stratix/default.htm

(6) 相田泰志；汎用評価ボードの製作，Design Wave Magazine，pp.66-75，2005年1月号．

S.Mori

<著者プロフィール>
家電メーカでIPコアの研究開発と主力製品への組み込みを担当しています．つい最近，ニキシー管時計を作るためにEAGLEを使ってOlimexで基板を起こしましたが，安くて便利なので電子工作（死語？）にはお勧めですね．

4